iEditors: Extending iTask with Interactive Plug-ins

Jan Martin Jansen[1], Rinus Plasmeijer[2], Pieter Koopman[2]

[1] Faculty of Military Sciences,
Netherlands Defence Academy, Den Helder, the Netherlands,
[2] Institute for Computing and Information Sciences (ICIS),
Radboud University Nijmegen, the Netherlands
`jm.jansen.04@nlda.nl, {rinus, pieter}@cs.ru.nl`

Abstract. The iTask library of Clean enables the user to specify web-enabled workflow systems on a high level of abstraction. Details like client-server communication, storage and retrieval of state information, HTML generation, and web form handling are all handled automatically. Using only standard HTML web browser elements also has a disadvantage: it does not offer the same level of interaction as we are used to from desktop applications. Browser plug-ins can fill this gap. They make it possible to extend web-applications with interactive functionality like the making of drawings. In this paper we explain how plug-ins can be nicely integrated in the iTask system. A special feature of the integration is the possibility for a plug-in to use Clean functions as call-back mechanism for the handling of events. These call-backs can be handled on the server as well as on the client. As a result we are now able to create interactive iTask applications (iEditors) using plug-ins like graphical editors. Although complicated, distributed multi-user applications can be created in this way, reasoning about the program remains easy since all code is generated from one and the same source: the high-level iTask specification in Clean.

1 Introduction

The internet has become an important platform for the deployment of applications. Despite this popularity, for an application programmer it is still hard to write web applications. To overcome this, the iData [18] and iTask [19] toolkits have been developed. They enable the development of web applications at a high level of abstraction, where the programmer can focus on the essence of the application without having to deal with web details like HTML generation and client-server communication. An iData application automatically generates output (HTML) and automatically handles user changes made in an HTML form. The iTask system adds the concept of tasks to iData. An iTask application can be considered as a structured collection of tasks to be performed by one or more users. In iTask specifications the flow of control and information between tasks can be expressed. To enhance the performance of iTask applications, the possibility to handle tasks at the client side of a web application was added. For this the SAPL interpreter [20] was extended to a full Clean interpreter [9].

iData and iTask make use of standard HTML elements. In many cases these standard elements do not suffice for the creation of desktop-like applications. Browser plug-ins can be used to overcome this. Examples of plug-ins are media players for playing music and movies and Java Applets that offer the possibility to run Java programs at the client side of web applications.

When developing a web application using a plug-in the programmer has to deal with the following issues:

1. How to include the plug-in in the web application?
2. How to load relevant data into the plug-in?
3. How to transfer relevant data from the plug-in to the server application?
4. How to do specific processing for the plug-in (e.g, event handling for editors)?

For the inclusion of plug-ins in web applications, standard solutions in HTML exist. The other issues are mostly handled on an ad hoc basis, depending on the kind of application developed.

In this paper we focus on a more systematic solution for the last three issues. The focus is on the inclusion of Java Applet plug-ins [6, 23] into iTask applications using generic [8] programming techniques. The presented techniques are not restricted to Java Applets alone but can also be used for communication with other kinds of plug-ins like advanced text editors (e.g. fckeditor [10]). For incorporating plug-ins into iTasks, a generic (read: poly-typical) framework is developed. The benefits for an application programmer are:

– A plug-in can be used with a minimum of programming effort and use of specific interface code. Generic functions take care of the conversion of Clean to Java data and back. They also take care of the communication between web-application and plug-in;
– One can define call-backs for the plug-in in Clean which can be handled either on server or client. Server handling can be used for executing more time consuming functions and client handling can be used for events requiring a quick response like mouse-event handling; For client side evaluation of call-backs the SAPL interpreter is used;
– Plug-in tasks behave like ordinary iTasks. If a suitable plug-in already exists, the application programmer only has to define Clean types (matching the content and event types of the plug-in), similarly to what is needed for ordinary iTasks. In order to include a plug-in into an iTask application only two interface functions are needed. For Java Applets the interface with plug-ins is encapsulated into a generic Java class.

We will call an iTask plug-in with Clean call-backs an iEditor. The technical contributions are:

– The seamless integration of plug-in tasks in the iTask formalism. This is realized by specializing the generic HTML generation and data update functions, in a completely transparent way for the application programmer;

– The use of Clean and SAPL dynamics [24] for realizing fine grained control over call-back function handling. Clean expressions are serialized at the server side, moved to the client side and executed there (this requires a referential transparent formalism). In this way it is possible to move entire computations from server to client in a dynamic way;
– A generic way to exchange data between Clean and Java. On the Clean side this is realized by standard generic print and parse functions. On the Java side this is realized by the Java reflection [14] mechanism.

The structure of this paper is as follows. In Section 2 we start with a short survey of the iTask system and architecture. In Section 3 we discuss the issues to be dealt with for including plug-ins in iTask and we give an example of the use of iEditors. Section 4 discusses the implementation of iEditors. In Section 5 we present a generic framework for the exchange of data between Clean and Java. In Section 6 we discuss some alternative uses and implementations of the techniques we developed. Section 7 compares our solution with other approaches that use client-side processing. Finally, we end with some concluding remarks in Section 8.

2 The iTask toolkit

The iTask toolkit [19] is a web-based combinator library written in the lazy, purely functional programming language Clean. It can be used to implement powerful web-applications like online shops, etc. We briefly repeat the most important characteristics of iTask. A task in iTask can be a basic task or a combination of tasks:

– A basic task is created by the `editTask` function, which turns an element of an arbitrary data type into an editable web form. User edits of the form lead to automatic updates of the underlying data type;
– Task combinators enable the combination of tasks. Combinators are used to control the flow of processing and data from one task to another. Tasks can be performed sequentially, in parallel and distributed over several users. New tasks can dynamically depend on the results of previous tasks.

In the original iTask architecture all processing is done at the server side of the application and all user actions lead to a complete update of the web-page the user is editing. In [20] we showed how we can update sub-tasks in web pages and reduce the overhead of client-server communication in iTask applications by adding Ajax [5] and client side evaluation of tasks. The plug-in extensions discussed in this paper extend the set of basic tasks with powerful interactive tasks. This is done in a way that does not restrict the way in which tasks can be combined with combinators.

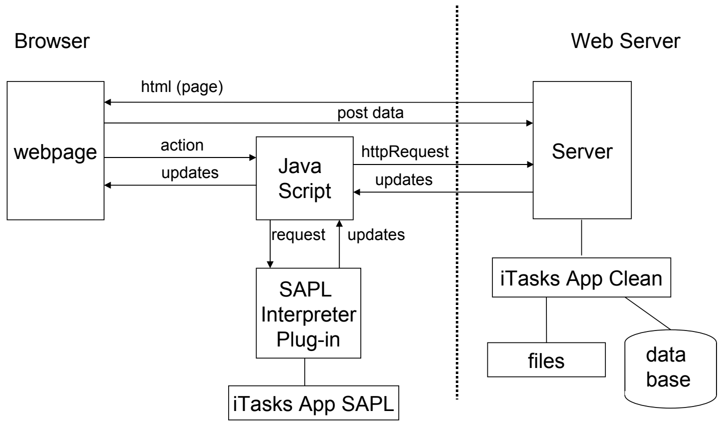


Fig. 1. The architecture of an iTask Application

2.1 The Architecture of iTask Applications

The architecture (see Fig. 1) of iTask applications is representative for web applications based on the Ajax philosophy (web 2.0 applications [5, 17], [25] gives details about web development with Ajax). It has the following characteristics:

- An iTask application consists of two images. A server executable running in native code at the server side of the application and a client side image running in the SAPL (Simple Application Programming Language) interpreter that is integrated in the web browser as a plug-in;
- Both the server and client images are generated from one single source programmed in Clean. From this source the server executable and a client SAPL program are generated by the Clean compiler. Both the Clean executable and the SAPL source comprise the complete iTask program. Tasks can be handled either at the server or the client. In principle, it is even possible to run the complete application (all tasks) at the client, except for the storage and retrieval of information in files and data bases;
- The server application initially generates a complete HTML page (web form) that is displayed in the client browser;
- User actions in the web form can be handled as normal post messages by the server or as an `httpRequest` by either client or server. In the first case a complete new HTML page is generated. In the second case it should be decided in JavaScript whether a request to either server or client application must be made. As result of the request a (partial) update of the web-page is made;
- The JavaScript at the client side is generic (the same for all iTask programs). JavaScript acts as an intermediary between client and server and client and SAPL interpreter. It takes care of updating the page with results from the server or client and it transforms user actions in the forms into calls for server or client application.

The use of JavaScript is a characteristic of all Ajax-based applications, but in our case the JavaScript functions are only a means for passing requests and results between the server application, client application and web-page. All application related programming is done in Clean. In this paper we extend this architecture with plug-in communication.

2.2 The SAPL Interpreter and Clean-SAPL dynamics

To execute tasks and Clean functions at the client-side, we need a Clean platform there. This is realized by making a plug-in version of the SAPL interpreter [9] and a Clean to SAPL compiler. By using a Java Applet for the interpreter, client-side Clean processing becomes available for all major internet browsers. The interpreter, originally realized in C, was re-implemented as a Java Applet with a performance penalty of less than 40%. This means that this interpreter is still considerably faster than other interpreters like GHCi, Helium and Hugs (see [9]). We also constructed a Clean to SAPL compiler, supporting the full Clean language. The generated SAPL code can be loaded into the SAPL interpreter at start-up of web applications. Loading times of SAPL and client program (excluding the time needed to load the Java virtual machine) are comparable to that of web pages including JavaScripts of about 1000 lines.

A special feature of the SAPL interpreter is that we can use a dedicated form of Clean dynamics [24] for it. With dynamics it is possible to serialize a Clean expression (closure) to a string, store the string somewhere, retrieve the string at a later moment, turn it into a Clean expression again and execute it. We extended the dynamics features of Clean in such a way that it is also possible to serialize an expression in a Clean executable and de-serialize it in the SAPL interpreter (running the corresponding SAPL program), and execute the expression there. This is a powerful feature because it makes it possible to migrate execution of a Clean program from server to client. In this paper we use this feature for executing call-back functions at the client side.

2.3 Examples of iTask Applications

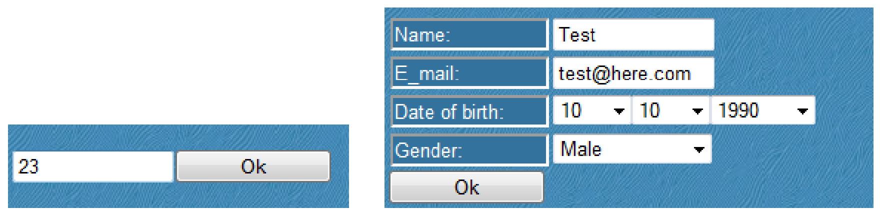
To give an idea of the iTask system, we give some small examples. Creating a basic task in iTask is simple. With the `editTask` function one can turn an element of an arbitrary data type into a task. As a result an editor for the data type element is created residing in a web form. A user edit action of this form results in an automatic update of the data type that can be further processed by the remainder of the iTask application. `editTask` has two arguments: the name of the button that the user should press to end the task and the initial value of the editor. Here two examples of the use of this function are given: `simpleInt` creates an editor for an integer while `simplePerson` creates an editor for an element of type `Person`. We also give the definition of the type `Person`.

```
simpleInt :: Task Int
simpleInt = editTask "Ok" createDefault
```

```
:: Person = { name        :: String
            , e_mail      :: String
            , dateOfBirth :: HtmlDate
            , gender      :: Gender
            }
:: Gender = Female | Male

simplePerson :: Task Person
simplePerson = editTask "Ok" createDefault
```

Fig. 2 shows the resulting editors created when respectively `simpleInt` and `simplePerson` are called. Note we use `createDefault` for the initial value of the



	23

Name:	Test
E_mail:	test@here.com
Date of birth:	10 10 1990
Gender:	Male

Fig. 2. editTask for Int (left) and Person (right)

editors. The fields in the form now get default values generated by the system using generic functions.

The ‘simple’ examples just create a form to be filled in by a single user, yielding a value of the corresponding type. In the following example a combinator is used to let two users perform tasks after each other:

```
addMultiUserTask :: Task Int
addMultiUserTask
=           0  @:: editTask "Ready" 0
  =>> λv → 1  @:: editTask "Ready" 0
  =>> λw → 0  @:: editTask "Result" (v+w),
```

User 0 (a login procedure binds a user to a unique id) has to enter a number, then user 1 has to enter a second number, then user 0 gets the sum of the numbers, but can still edit the result.

`=>>` is the iTask equivalent of the monadic ‘bind’ operator. `n @:: task` assigns `task` to user `n`.

3 iEditor: Plug-ins in iTask

Plug-ins are used for features that are not supported by standard HTML constructs like interactive drawing, complex text editing and animations. Plug-ins have to be installed by the user of the browser. Once this is done, they can be loaded by special HTML constructs. The use of plug-ins however, complicates the development of web applications. The developer has to take care that the plug-in is initialized and that the data needed by the plug-in is passed to it. In some cases, data from the plug-in has to be passed back to the web application or events occurring in the plug-in have to be handled by the web applications (e.g. mouse events).

In this section we introduce iEditor, an extension to iTask for the integration of plug-ins and give an example of its use.

3.1 The iTask architecture including iEditors

In Fig. 3 we show the adapted iTask architecture for including iEditors. The

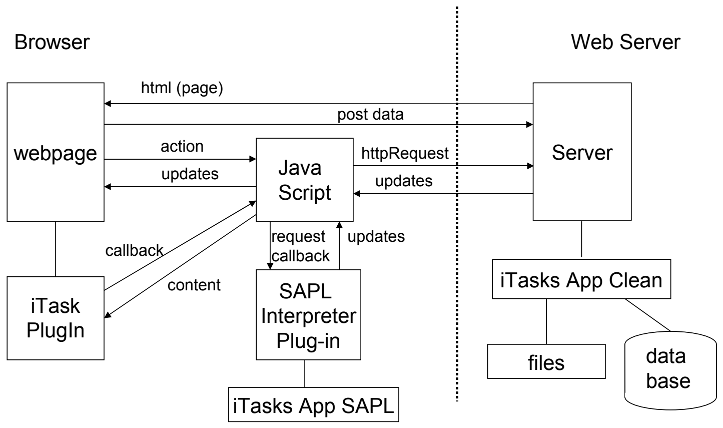


Fig. 3. The architecture of iTask with plug-ins

extensions with respect to the standard iTask architecture (Fig. 1) are:

- The plug-in is part of a web-page. This means that the initial web-page should contain an HTML representation of the plug-in;
- All communication with a plug-in must be done via JavaScript functions. This is the standard way of communication with plug-ins. All popular plug-ins can be accessed from JavaScript and can call JavaScript functions. Although it is possible to communicate directly with Java Applets from the SAPL interpreter, we use the indirection via JavaScript to obtain a uniform interface that can also be used for non-Java plug-ins;
- For call-backs from the plug-in, the JavaScript function handling them has to decide where the calls should be made: either server or client.

3.2 The `PlugIn` wrapper type

In a basic iTask an element of a data type is turned into an editable HTML form by the function `editTask` and the result of editing the form is automatically turned into an updated instance of this data type. We want to maintain this interface for iEditors. More concretely, the information exchanged with a plug-in must also be represented by a data type and the use of the plug-in should lead to an updated instance of this data type. Because `editTask` has no means of distinguishing a data type intended for a plug-in from any other data type and also because we need information about how to load and display the plug-in, we have to wrap the content data type into a special `PlugIn` data type. For this wrapper type we can now make a specific implementation of the `editTask` function.

```
::PlugIn ct et st = {plugininfo    :: PlugInInfo,
                     content       :: ct,
                     events        :: [et],
                     state         :: st,
                     callback      :: [et] (ct,st) → (ct,st),
                     isServerEvent :: et → Bool}
```

The wrapper type contains all information needed for the creation of the plug-in (the right HTML code). It also contains all information needed to enable communication from plug-in to JavaScript and vice versa. `PlugIn` has three type parameters `ct`, `et` and `st`:

- `ct` is the type of the content to be exchanged with the plug-in;
- `et` is the type of the events that can occur in the plug-in;
- `st` is the type of the state that must be maintained between calls of the call-back. This type is not visible to the plug-in itself, but only to the call-back function that handles events from the plug-in.

The fields in the PlugIn type have the following meaning:

- `plugininfo`: information for constructing the HTML representation of the plug-in: how to load the plug-in, its size and other initializing parameters (see the example in Section 3.4);
- `content`: content of the plug-in. This field contains the initial content of the plug-in and after the plug-in is ready it contains the result of the plug-in;
- `events`: generated events that have to be processed by the call-back function;
- `state`: value of the state to be maintained between call-back calls;
- `callback`: call-back function that handles the generated events;
- `isServerEvent`: indication where events have to be handled.

The call-back function takes the generated events, the current content and state as input and returns a new content and state. The content is passed back to the plug-in. The state is maintained for the next call of the call-back. The call-back function is automatically called from the plug-in whenever an event occurs. On the plug-in side there should be data types where the content and event types can be mapped on (more details in Section 4). Mismatches will lead to the generation of exceptions on either Clean or plug-in side.

For indicating where events have to be handled, the user must specify the function `isServerEvent`. If this function returns `True` for an event, this event is handled on the server; if it returns `False`, the event is handled on the client.

From the iTask point of view an iEditor is just another editor for a data type (the content field). All other information in the PlugIn type is only there for enabling the creation of the iEditor and for doing processing (event handling) for the plug-in (invisible at the iTask level). For plug-ins not requiring event processing, the events, state and callback fields can be filled with stubs.

3.3 Interface functions for a Plug-in

For exchanging information between the iTask program and the plug-in two interface functions (one for the plug-in and one for JavaScript) are needed:

```
setContent(String content)
doPlugInCall(String pluginid, String content, String events)
```

setContent should be implemented by the plug-in and must be callable from JavaScript. doPlugInCall is a JavaScript function and must be called by the plug-in. pluginid is a unique id, identifying the plug-in (there can be more than one plug-in). The content and events arguments are serialized versions of the corresponding Clean datatypes (see Sections 3.4 and 4). For Java Applets we provide a generic Java class that takes care of the communication between plug-in and iTask program (see Section 5).

For other plug-ins there are two possibilities. Either the plug-in should be adapted by wrapping code that supports these functions, or special interface code can be written in JavaScript taking care of the conversion of Clean data to data compatible with that of the plug-in. Often, this interface code can be used for a whole class of similar plug-ins.

3.4 A Graphical Editor Plug-in for iTask

We now look at an example of the inclusion of an iEditor in iTask: a simple graphical editor. We assume, we have created a Java Applet plug-in that is capable of displaying simple graphics (lines, ovals, rectangles, etc.) and that can generate events for mouse and button actions. The processing of events depends on what kind of graphical editor we want to make (vector graphics, diagrams, etc.). It is possible to create a dedicated plug-in for each kind of editor, but by using Clean for doing event handling we can adapt the behavior of the application by only changing the Clean source, without the need to adapt the plug-in. The key idea is that mouse and button events are passed to the web-application by a call-back function call. The call-back function can either be executed on the server by the Clean executable or on the client by the SAPL version of the Clean application. For each type of event, the programmer can choose where it must be handled.

We give the (almost) complete Clean source code for this editor. We start with the data types:

```
::GraphObject  = GraphLine Int Int Int Int | GraphOval Int Int Int Int |
                 GraphRect Int Int Int Int | GraphPolyLine [Pnt]       |
                 GraphButton String
```

```
::Pnt          = Pnt Int Int

::GraphEvent   = MouseDown Int Int | MouseDrag Int Int |
                 MouseUp   Int Int | ButtonEvent String

::GraphState   = NewLine |  NewPolyLine | NewRect | NewOval
```

In the application a drawing is represented by a list of `GraphObject`. We distinguish several types of figures and simple buttons (for the sake of simplicity we combined figures and buttons in one type). `GraphEvent` represents the events that can occur. We distinguish mouse (down, up, drag) and button events. The `Int`s represent the `x` and `y` position of the mouse event We assume that the plug-in is capable of displaying elements of `GraphObject` and that it turns events into elements of `GraphEvent`. The plug-in should have matching types for `GraphObject` and `GraphEvent`. The transformation of elements of these types onto each other is done automatically (see Section 4 and 5). `GraphState` is a state data type maintaining that part of the state that is not passed to the plug-in, but that is needed by the call-back function. In this example it maintains the type of the figure to be drawn at a mouse down event.

The task definition is given by:

```
graphtask ::  Task (PlugIn [GraphObject] GraphEvent GraphState)
graphtask =   editTask "Ready" graphplugin
```

The initialization of the plug-in is given by:

```
graphplugin :: PlugIn [GraphObject] GraphEvent GraphState
graphplugin =  {plugininfo    = grapheditapplet,
                content       = initpicture,
                events        = [] ,
                state         = NewLine,
                callback      = doEvents,
                isServerEvent = isMouseUp}

isMouseUp (MouseUp _ _) = True
isMouseUp            _  = False

grapheditapplet = AppletPlugIn {id      = "drawplugin",
                                archive = "drawapplet.jar",
                                code    = "drawapplet/maincanvas.class",
                                width   = 500,
                                height  = 200}

initpicture = [GraphButton "Line",      GraphButton "PolyLine",
               GraphButton "Rectangle", GraphButton "Oval"]
```

`graphplugin` contains the initialization of the plug-in. We see that all events are handled on the client except `MouseUp` events. As a consequence, the server side `PlugIn` data type is updated at every `MouseUp`. `grapheditapplet` contains the information needed for generating the HTML representation of the plug-in:

the Applet id, the codebase and main class, its width and height. Finally, we see that the initial picture only contains the buttons.

Events occurring in the plug-in, are handled by the `doEvents` function:

```
doEvents ::    [GraphEvent] ([GraphObject], GraphState) →
                            ([GraphObject], GraphState)

doEvents [ButtonEvent "Line":evs]   (figs,_)
= doEvents                  evs     (figs,NewLine)

doEvents [MouseDown x y:evs]        (figs,NewLine)
= doEvents                evs       ([GraphLine x y x y:figs],NewLine)

doEvents [MouseDrag x y:evs]        ([GraphLine v w _ _: figs],a)
= doEvents                evs       ([GraphLine v w x y: figs],a)

doEvents [e:evs] (figs,a) = doEvents evs (figs,a) // ignore other events
doEvents []      (figs,a) =              (figs,a) // return result
```

Here only the code for `Line` is shown, `Rectangle`, `PolyLine` and `Oval` are handled in a similar way. Fig. 4 shows a screen shot of the application.

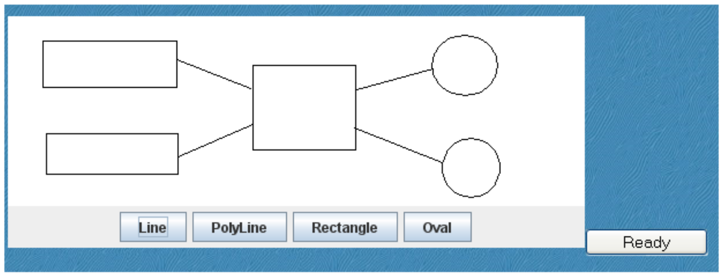


Fig. 4. Screen shot of the drawing application

The user can stop editing by clicking the 'Ready' button. The current content and state are now made available to the remainder of the iTask application.

A multi-user graphical editor To show that the plug-in task simply behaves like a normal iTask we give a small variation of `graphtask` analogous to the multi-user example from Section 3:

```
graphtask ::  Task (PlugIn [GraphObject] GraphEvent GraphState)
graphtask =             0 @:: editTask "0 Ready" graphplugin
              =>> λv → 1 @:: editTask "1 Ready" v
              =>> λw → 0 @:: editTask "Result" w
```

Two users are involved in this example. User 0 makes an initial drawing. The result is passed to user 1, who can further edit the drawing. If this task is ready the result is passed back to user 0 who can continue editing.

For this application the programmer only has to specify the (content, state and event) types and the call-back function needed for handling events. The plug-in iTask behaves like an ordinary iTask. All communication with the plug-in is handled in a way that is transparent for the programmer.

It is also possible to wrap several plug-ins into one task. For example: `editTask "Ready" (graphplugin,texteditplugin)` wraps two editors together into one form. The editors are displayed next to each other.

4 Implementation of iEditors

From the example it is clear that the use of iEditors is straightforward for the application programmer and that, from the iTask point of view, an iEditor is just another editor. In the implementation we face a number of challenges (to answer questions 1 to 4 from Section 1):

- How to fit the plug-in into the iData/iTask architecture?
- How to exchange data between server, plug-in and client?
- How to invoke call-back functions from plug-in for server and client?

4.1 Fitting a plug-in into the iTask architecture

The HTML representation of the plug-in is generated as part of the initial iTask web-page. This is realized by making a specialized implementation of the generic `gForm` [18] function that is part of the implementation of `editTask` and that is responsible for the generation of the web form. The resulting HTML also contains the initial content of the plug-in and all other information needed by the plug-in and by the JavaScript functions that interact with the plug-in. This adapted `gForm` is generic and works for all plug-ins.

4.2 Data exchange between client, plug-in and server

For data exchange between plug-in, server and client Clean program we use the generic print and parse functions on the Clean side (on server and client). As a consequence the plug-in must have a (generic) way to parse and unparse the strings representing the event and content data types. In Section 5 we discuss how this is realized for Java.

Although we have the same Clean program running at both the server and client side, the internal representations of data types are completely different. Therefore we also use the generic print and parse functions for the exchange of data between the Clean programs at server and client side.

All communication between server, client and plug-in is done using JavaScript functions, similar to what is done for Ajax and client side handling of iTask tasks (see [20]). These functions are generic in the sense that they do not depend on the specific plug-in. The JavaScript functions are responsible for passing data from plug-in to server and client and vice-versa, but also for making the call-back and deciding where the call-back must be handled. The addition of plug-ins only requires one extra JavaScript function to handle all communication from the plug-in with client and server Clean programs:

```
doPlugInCall(String pluginid, String content, String events)
```

This function can both handle the final result from a plug-in and the call-backs generated by the plug-in. The first argument is the unique ID of the plug-in (there can be more than one plug-in and they all use this JavaScript function). The second argument is the serialized version of the current content of the plug-in. The third argument is a serialized version of the list of the events that must be processed (this list is empty in case the plug-in just wants to synchronize its content with the server program). The JavaScript function takes care of either updating the server program with the content of the plug-in or by making the call-back to client or server program (see Section 4.3).

For updating its content the plug-in should implement the following function:

```
setContent(String content)
```

The argument is again a (serialized) string representation of the content. This function is called from JavaScript. It is custom for plug-ins to support function calls from JavaScript.

4.3 Handling call-backs

Call-backs can be made to either client or server. The plug-in makes the call-back by calling the (generic) Javascript function `doPlugInCall` with the serialized content and events as arguments. The JavaScript function determines whether the call must be handled on the server or the client by executing the `isServerEvent` function for the event in the SAPL interpreter. For the server case, the `PlugIn` data type is updated in a similar way as for an ordinary update for iData [18]. The implementation of `editTask` makes use of the generic function `gUpd` for updating the data type with the result of a user edit action. For the `PlugIn` data type a specialized version of `gUpd` is made that applies the call-back function to its arguments before updating the `PlugIn` data structure with the result. Finally, the HTML representation of the plug-in is re-generated with the new content.

We could use the same strategy for the client side (the full Clean program is available). But we must do it in a much more efficient way, because the overhead of finding out for which task the update is intended can be large. We can do it more efficiently because we have an interpreter available that can execute an arbitrary Clean expression. We use this to directly execute the call-back function call and use the result to make a direct update of the content of the plug-in. In this way we short-circuit the use of `gUpd` and the whole iTask machinery needed for finding out which task is updated [19]. This optimization is absolutely necessary for events needing immediate response like mouse drag events.

For making the direct call-back on the client we use Clean-SAPL dynamics (see Section 2.2). For this, the serialized call-back function is stored in the plug-in HTML representation. Not only is the call-back function itself serialized, but the `isServerEvent` function and the parse and unparse functions for the arguments (content, state and events) are also serialized. The last is necessary because the arguments are passed to the call-back as serialized strings from the plug-in via the `doPlugInCall` function and the result must be passed back in serialized form too.

In the actual call-back, it is first checked if the event is really intended for the client by applying the `isServerEvent` function to the deserialized event. If not, the server call-back is made as described above. Otherwise the call-back and parse and unparse functions are all de-serialized, the arguments are parsed, the call-back is applied, and the result state and content are unparsed again. The content is handed back to the plug-in directly (via `setContent`) and the state is maintained in the HTML representation of the plug-in.

Note that we cannot handle all call-backs at the client side. Processing intensive call-backs and call-backs requiring information from data bases or files should be handled on the server side.

4.4 Evaluation of Efficiency of handling call-backs

In the graphical editor application we used the call-back function to handle mouse down and drag events by the SAPL interpreter. Mouse drag events often occur in quick sequences (in the order of 10-15 events per second). The whole call-back machinery was capable of keeping track of these events on an Intel 1.6 GHz Core Duo 2 machine (using only one core). Attempts to handle the drag events by the server lead to a browser hang-up due to a client-server communication overload. Of course, the (de)serialization of data types takes a significant amount of time and is a limiting factor in the amount of events that can be handled. Native implementations (without the need to (de)serialize) can easily handle up to ten times as much events.

5 Implementation for Java Applets

Java Applets [6, 23] are an important class of plug-ins. All modern web browsers offer the possibility of Applet plug-ins. In this way it is possible to incorporate complex Java applications into web pages. We already used the Java Applet mechanism for loading the SAPL interpreter at the client side of iTask for handling client tasks and call-backs. Although Java Applets can offer rich functionality they are less popular, because communicating with them must be handled in an ad-hoc manner, making it difficult to integrate them with the remainder of a web application (see also [21]). By using the iTask plug-in techniques, we have a generic strategy which simplifies the communication with Java Applets. To include a Java Applet in an iTask application we have to deal with the following issues:

– We have to find a way to map Clean types on corresponding Java data types;
– We have to take care that we offer the interface needed for communication with JavaScript.

5.1 Mapping Clean and Java Data Types onto each other

In order to exchange information between a Clean and Java application there must be a way to transfer Clean data to Java data and back. To save the programmer from writing boilerplate data transformation code we included generic code in Clean and Java to handle this data transformation. Not all Clean and Java data types can be mapped onto each other. For a Java class the member fields are (currently) restricted to the following types:

– primitive types: (`int`,`long`,`float`,`double`,`boolean`,`char`);
– the `String` type;
– all subtypes of `List` (they are all mapped on a Clean list);
– other Java classes with members that obey these rules.

A class may be a subclass of another class or implement an interface, but all superclasses must obey the rules mentioned above. Other (container) types, like `Map` and arrays are not (yet) allowed. For these classes an ad-hoc mapping must be made (like is done for `List`).

From the Clean point of view, the automatic conversion of Clean data types to Java types is restricted to first order data types that can be described by standard Algebraic Data Types. Records are not allowed yet, but they can be easily added. If a Clean type is mapped onto a Java type hierarchy the fields of the Clean type should match the union of all fields in the class hierarchy in the order of the hierarchy (fields of superclass before fields of subclass).

More formally, consider the following algebraic data type definition in Clean:

```
::typename t1 .. tk = C1 t11 .. t1n_1 | .. | Cm tm1 .. tmn_m
```

`t1..tk` are type parameters, `C1..Cm` constructor names and `tik` type names or type parameters. This type definition corresponds to the following Java interface and `m` Java classes:

```
interface typename {}
class<t1,..,tk> C1 implements typename {t11 a11; .. t1n_1 a1n_1;}
...
class<t1,..,tk> Cm implements typename {tm1 am1; .. tmn_m amn_m;}
```

Each constructor is represented by a separate Java class with as name the constructor name and with as fields the arguments of the constructor (with names `aik`) with their type. As an example, the Clean type `GraphObject` from Section 3 corresponds to the following Java classes:

```
interface GraphObject {}
class GraphLine implements GraphObject {int x, y, v, w;}
class GraphRect implements GraphObject {int x, y, v, w;}
class GraphOval implements GraphObject {int x, y, v, w;}
class GraphPolyLine implements GraphObject {List<Pnt> points;}
class GraphButton implements GraphObject {String name;}

class Pnt {int x, y;}
```

It is possible to generate a corresponding Clean data type definition from an existing Java class (hierarchy) using generic Java functions. Otherwise the programmer has to take care that matching types at Clean and Java side exist, as we did in our graphics editor example. Once corresponding data types exist, the conversion of data is done automatically.

For the actual conversion of data we use the standard generic print and parse functions at the Clean side (`gPrint` and `gParse`) and reflection [14] on the Java side.

5.2 Java Applet Plug-In Interface

To further simplify the communication between Java plug-in and Clean iTask application, the Java class `CleanJavaCom` is offered. This class contains member functions that can be used for parsing and unparsing Java objects and functions for handling the communication with the JavaScript interface. The `CleanJavaCom` class is generic and can be used in every Java Applet to be used as plug-in in an iTask application.

```
class CleanJavaCom<CT,ET> {
  private String writeClassToString(Object object)  {...}
  private Object readClassFromString(String inp)    {...}
  public CT getContent() {...}
  public void setContent(String ser_content) {...}
  public void handleEvents(List<ET> events) {...}
  ...
}
```

The class is parametrised by the Java versions of the content (`CT`) and event (`ET`) types.

- `writeClassToString` generates a string representation of an object that exactly fits the Clean representation;
- `readClassFromString` parses a string representation generated by `gPrint` to the corresponding Java object;
- `getContent` is called to obtain the content by the remainder of the Applet;
- `setContent` is called from JavaScript to set the content. The content string is de-serialized by a call to `readClassFromString`. The result can now be obtained by the remainder of the applet by a call of `getContent`;
- `handleEvents` is called by the Applet after one (or more) event(s) have occurred. This function takes care of serializing the content and events and calling the `doPlugInCall` function in JavaScript. JavaScript should pass back the result by a call of `setContent`. If the plug-in only wants to synchronize its content with the iTask server application it should call `doEvents` with an empty event list.

In the implementation of `writeClassToString` and `readClassFromString` the Java reflection mechanism is used.

6 Discussion

Plug-ins must have matching types for the content and event types. For Java we implemented a generic way to convert the serialized content and event types to Java data structures and back. Not all plug-in types offer the possibility to do this conversion in a generic way. An alternative is to use generic functions in Clean for generating a representation the plug-in can deal with and for parsing back the results. An example is the use of XML [2]. Java has the `XMLEncoder` and `XMLDecoder` classes for generating and parsing XML representations of data types. For us, a more interesting alternative is the use of JSON (JavaScript Object Notation) [1]. This has as an advantage that we can also exchange data with JavaScript and a large number of other formalisms. Like string serialization, it allows for a lightweight implementation with little overhead. We already started to implement generic generation and parsing of JSON data in Clean and we will use this for future implementations.

Other alternatives are the use of CORBA [16] or the Java Native Interface [12] for exchanging data between Clean and the plug-in. Examples can be found in [15, 21, 4]. For us, these approaches are too heavyweight to be used at the client side in the SAPL interpreter.

The idea of attaching an event handler to an editor is not restricted to plug-in tasks. Call-back functions can also be attached to other basic iTask editors. The call-back can be used to check the content of the editor before sending it back to the server and give the user feedback in case something is wrong or to reformat the content before displaying it again. Because the full power of Clean is available at the client side there are no restrictions to the call-back functions that can be defined. In this way, the concept of iEditors can be extended to arbitrary iTask editors. To implement this, we can either use a wrapper type like `PlugIn` or introduce a special combinator in iTask like the one used for assigning users to tasks (see Section 2).

7 Related Work

In this paper we extended the iTask toolkit with a generic framework for the inclusion of plug-ins, with the possibility to make calls from the plug-in to Clean functions that can be executed on either client or server. We are not aware of any other functional system that has these features. However, there are functional approaches for handling web pages using the same formalism for server and client side processing. Most of them compile to JavaScript for client side execution. An example of this approach is Hop [22, 13]. Hop is a dedicated web programming language and its syntax is HTML-like. In Hop it is also possible to specify a complete web application without the (direct) use of JavaScript. Hop uses two compilers, one for compiling the server side program and one for compiling the client-side part. The client side part is only used for executing the user interface. The application essentially runs on the client and may call services on the server. Hop uses syntactic constructions for indicating client and server part code. It is build on top of the Scheme programming language. In our case we do not have to extend Clean, but can write the entire web application in Clean itself. In [13] it is shown that a reasonably good performance for the client side functions in Hop can be obtained. For us, compiling to JavaScript is no option because Clean is lazy. Instead we use the SAPL interpreter, which also has competitive performance as was shown in [9] and the graphics editor application.

Links [3] and its extension formlets is a functional language-based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. A Links program stores its session state at the client side. In a Links program, the keywords `client` and `server` force a top-level function to be executed at the client or server respectively. In Links, processes can be spawned and these processes can communicate via message passing. Client-server communication is implemented using Ajax technology, like we do. In the iData and iTask toolkits, forms are generated generically for every data type, whereas in Links and Formlets these need to be coded by the programmer.

The Flapjax language [11] is an implementation of functional reactive programming in JavaScript, with features comparable to those of Hop. Both are designed to create intricate web applications. In Flapjax, Hop and Formlets processing is directly attached to web form handling, which is comparable to the use of call-backs in iEditors.

A much more restricted approach has been implemented in Curry [7]: only a very restricted subset of Curry is translated to JavaScript to handle client side verification code fragments only.

Summarizing the main differences with the other approaches are:

– iTask/iEditor applications are just plain Clean applications, where web forms are generated from data types. The other approaches define dedicated web languages where processing is attached to web forms;
– We can use the full Clean functionality at the client side because the SAPL interpreter offers a full Clean platform. The other approaches rely on compilation to JavaScript with, in many cases, restrictions on the functions that can be compiled to JavaScript;
– Clean-SAPL dynamics offers a generic and flexible way to attach call-back handling to web forms and plug-ins. Where the other approaches use static annotations to indicate whether functions have to be executed on either client or server, in our approach this can be decided dynamically, depending on the events to be processed.

8 Conclusions

Plug-ins are often an essential part of more interactive web applications. In this paper we discussed a generic way for including plug-ins in iTask applications. All communication between iTask application and plug-in is on the level of exchanging and updating data types, which is entirely consistent with the normal way iTask works. Plug-in tasks behave like ordinary tasks. No adaptations of iTask were necessary to incorporate them, only a specialization of the `gForm` and `gUpd` functions for the PlugIn type.

An important feature is that plug-ins can use Clean functions, which can be executed on either server or client, for event handling. This gives the programmer fine-grained control over the behavior of the plug-in without the need to adapt the plug-in itself. In this way, we can keep the plug-in to its essence and use Clean for all processing not involving the specialities of the plug-in.

Information exchange between server, client and plug-in is realized with the use of generic (un)parsing of data types. For efficient client side event handling a combination of Clean-SAPL dynamics and generic (un)parsing is used. With Clean-SAPL dynamics it is possible to move the execution of arbitrary Clean expressions from server to client. This turns out to be a powerful feature that can also be used for attaching client side functions to arbitrary web forms.

For Java Applets, a straightforward to use generic class is provided that handles all interaction of the plug-in with Clean including the conversion of data types and the forwarding of call-backs. Plug-ins of other type should implement a simple JavaScript interface and the (de)serialization of the data types used for the exchange of information.

We have maintained the declarative approach of the iTask toolkit. Server and client programs and all call-back handling functions are generated from an annotated, single-source specification with a low burden on the programmer because the system itself switches automatically between client and server side evaluation of tasks and call-backs when this is necessary. The iTask system integrates all mentioned technologies in a truly transparent and declarative way.

References

1. *Introducing JSON.* www.json.org, visited March 2009.
2. T. Bray, J. Paoli, and C. Sperberg-Macqueen. Extensible Markup Language (XML) 1.0 (w3c recommendation), 1998. Technical Report, http://www.w3.org/TR/1998/REC-xml-19980210.
3. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects, FMCO '06*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.
4. E. Evans and D. Rogers. Using Java applets and CORBA for multi-user distributed applications. *Internet Computing, IEEE*, 1:43–55, 1997.
5. J. Garrett. Ajax: A new approach to web applications, 2005. www.adaptivepath.com/ideas/essays/archives/000385.php, visited March 2009.
6. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Sun Microsystems, 1996.
7. M. Hanus. Putting declarative programming into the web: Translating Curry to JavaScript. In *Proc. of the 9th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'07)*, pages 155–166. ACM Press, 2007.

8. R. Hinze. A new approach to generic functional programming. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2000.
9. J. M. Jansen, P. Koopman, and R. Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In H. Nilsson, editor, *Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, UK, 19-21 April 2006, The University of Nottingham*, volume 7 of *Trends in Functional Programming*. Intellect Publisher, 2006.
10. F. C. Knabben. FCK editor, 2003. www.fckeditor.net, visited March 2009.
11. S. Krishnamurthi. The Flapjax site, 2007. www.flapjax-lang.org, visited March 2009.
12. S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Company, 1999.
13. F. Loitsch and M. Serrano. Hop client-side compilation. In *Trends in Functional Programming, TFP 2007, New York*, pages 141–158. Interact, 2008.
14. G. McCluskey. Using Java reflection, 1998. http://java.sun.com/developer/technicalArticles/ALT/Reflection/index.html, visited March 2009.
15. E. Meijer and S. Finne. Lambada: Haskell as a better Java. In *Proceedings of the 2000 Haskell Workshop, Montreal, Canada*, 2000.
16. OMG: Object Management Group. *The Common Object Request Broker: Architecture and SpecificationRevision 2.0*, 1996. http://www.omg.org/corba-e/index.htm, visited March 2009.
17. L. D. Paulson. Building Rich Web Applications with Ajax. *Computer*, 38(10):14–17, 2005.
18. R. Plasmeijer and P. Achten. The implementation of iData. In A. Butterfield, C. Grelck, and F. Huch, editors, *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Dublin, Ireland, September 19-21, 2005, Revised Selected Papers*, volume 4015 of *Lecture Notes in Computer Science*, pages 106–123. Springer, 2006.
19. R. Plasmeijer, P. Achten, and P. Koopman. iTasks: Executable specifications of interactive work flow systems for the web. In N. Ramsey, editor, *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming, Freiburg, Germany, October 1-3, 2007*, volume ICFP'07 of *International Conference on Functional Programming*, pages 141–152. ACM, 2007.
20. R. Plasmeijer, J. M. Jansen, P. Koopman, and P. Achten. Declarative Ajax and client side evaluation of workflows using iTasks. In *Principles and Practice of Declarative Programming, Valencia, Spain, July 2008*, volume PPDP 08, 2008.
21. C. Reinke. Towards a Haskell/Java connection. In *Implementation of Functional Languages, IFL 1998*, volume 1595 of *Lecture Notes in Computer Science*, pages 203–219. Springer, 1998.
22. M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006), Portland, Oregon, USA, October 22-26 2006*, pages 975–985, 2006.
23. Sun Microsystems. Release notes for the next-generation Java Plug-In technology, 2008. jdk6.dev.java.net/plugin2, visited March 2009.
24. A. van Weelden. *Putting Types To Good Use*. PhD thesis, Radboud University Nijmegen, the Netherlands, 2007.
25. W3 Schools. Ajax tutorial, 2008. http://w3schools.com/ajax/default.asp.